Pitfalls and tips when using Coco/R

The following examples and discussion highlight the various features of Cocol, and draw attention to common beginners' problems and misconceptions. There is a full "user manual" for Coco/R in the text book, chapter 10. There is also a complete, comprehensive, Coco Manual on PDF format available on the course web page at `http://www.cs.ru.ac.za/courses/CSc301/Translators/CocoManual.pdf`.

CHARACTERS section

This *optional* section describes the character sets from which tokens can be built in the later TOKENS section. It does nothing more; in particular it does not define tokens themselves even though at first it might seem to do so. While the definitions seem to introduce strings, this is illusory - the quotation marks are used in the sense that mathematicians would use curly braces.

Any number of sets can be defined, and they can have elements in common or not, as is convenient.

The + and - operators can be used to form the union or differences of sets, and the .. notation can be used to simplify the specification of a range. This is especially useful where this range includes control characters.

Avoid putting escape sequences like \n into the strings that define sets. Rather use the CHR(n) notation.

One cannot put spaces into literal strings in Coco/R (see later), whether these strings are being used in the CHARACTERS section to denote a set of characters, or in the TOKENS or PRODUCTIONS section to denote a literal string (terminal).

From these two points it follows that control characters like CR (CHR(13)) and LF (CHR(10)) have to be represented using the CHR(n) notation. So does SP (space, or CHR(32)) if for some strange reason it is needed as an *internal* character in a token.

`ANY` is a keyword that in this context means "all characters acceptable to the implementation of Coco/R". Although Java itself uses Unicode, Coco/R for Java is restricted at present to the low end (8 bit) subset of UNICODE. The first 128 elements of ASCII and Unicode are the same.

Examples

Here are some correct CHARACTERS declarations:

```
  CHARACTERS
    ULetter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .  /* set of upper case letters */
    LLetter    = "abcdefghijklmnopqrstuvwxyz" .  /* set of lower case letters */
    digit      = "0123456789" .                  /* set of all possible digits */
    hexDigit   = "0123456789ABCDEF" .            /* Hexadecimal digits */
    AnyVowel   = "AEIOUaeiou" .                  /* The English vowels in either case */
```

Here are some alternatives that are valid. We can use ranges, set unions and differences:

```
    ULetter    = "A" .. "Z" .                    /* set of upper case letters */
    LLetter    = "a" .. "z" .                    /* set of lower case letters */
    letter     = ULetters + LLetters .           /* set of all possible letters */
    control    = CHR(0) .. CHR(31) .             /* set of all ASCII control characters */
    inString   = ANY - control - '"' - CHR(92) . /* set of all easy characters within a string */
    printable  = ANY - control .                 /* set of all easily printable characters */
    hexDigit   = digit + "ABCDEFabcdef" .        /* all hex digits, either case acceptable */
```

Here are some "singleton" ones used to get round the problem of representing certain characters in the "string" version:

```
    LF         = CHR(10) .    /* Line feed */
    CR         = CHR(13) .    /* Carriage return */
    backslash  = CHR(92) .    /* the dreaded \ used in escape sequences */
    Space      = CHR(32).
    WhiteSpace = CHR(9) .. CHR(13) + Space .
```

Here are some invalid ones:

```
    BadLetter = "A..Z" .   /* bad notation */
    BadDigit1 = { "0" , "1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9" } . /* bad notation */
    BadDigit2 = { "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" } . /* bad notation */
    BadDigit3 = { "0123456789" } .                                              /* bad notation */
    BadDigit3 = { 0 1 2 3 4 5 6 7 8 9   } .                                     /* bad notation */
    CRLF      = CR LF  .   /* cannot define a TOKEN in this section */
```

You cannot put the names of CHARACTER sets, or the notations using CHR into the PRODUCTIONS, no matter how tempting this might seem:

```
  CHARACTERS
    CR = CHR(13);
  ...
  PRODUCTIONS
    Line = "SOMETHING" /* Here the SOMETHING is a string representing a TOKEN, and is allowed */
           CR          /* Wrong - CR is the name of a singleton CHARACTER set, not a token */
           CHR(10) .   /* Wrong  - CHR(n) can only be used in defining CHARACTER sets or TOKENS */
```

TOKENS section

The *optional* TOKENS section is typically used to define the expected patterns that things like identifiers, strings and numbers can have. That is, it specifies how generic tokens may be constructed. "Generic" tokens are effectively regarded by the parser as terminal symbols, of which there might be several instances. They are all described in the same way, but they are actually lexically distinct (because they will actually be spelled differently).

The definitions are done using the Wirth EBNF metabrackets, but restricted effectively to rules equivalent to regular expressions. Any names that appear in these definitions must be the names of CHARACTER sets, and effectively here mean "select any character from that character set". Here are the obvious common ones:

```
  TOKENS
    identifier     = letter { letter | digit } .  /* For example Pat or Terry */
    unsignednumber = digit { digit } .            /* For example 1234 or 51 */
    CRLF           = CR LF .
```

One can also define tokens by a mixture of such characters drawn one at a time from named character sets and explicit strings. Within a token definition these strings "stand for themselves". Here are some simple examples:

```
  TOKENS
    signedNumber   = ( "+" | "-" ) digit { digit } .  /* mandatory + or - followed by digit sequence */
    hexNumber      = digit { hexDigit } "H" .
    string         = '"' { instring | backslash printable } '"' .
```

One can declare complete, very specific tokens in this way too. For example

```
  TOKENS
    BeginKeyWord = "begin" .
```

in which the string "`begin`" stands for itself. This is unusual - keywords are nearly always declared simply by strings that appear where needed in the PRODUCTIONS section.

Tokens can get quite involved. For example, consider the following:

```
  TOKENS
    floatnumber    = [ "+" | "-" ] digit { digit }    /* whole part */
                     "." digit { digit }              /* digits after the mandatory point */
                     [ ( "E" | "e" )                  /* optional exponent part */
                       [ "+" | "-" ]                  /* optional sign within this */
                       digit { digit }                /* mandatory digit sequence */
                     ] .
```

This definition demands that there must be at least one digit before the point, there must be a point, and at least one digit after the point.

Convince yourself that this could match numbers like +12.3 or 12.4E+5 or 12.4E3 or 1.4e26.

Here is a variation that will match integer or float numbers:

```
  TOKENS
    floatnumber    = [ "+" | "-" ] digit { digit }      /* whole part */
                     [                                  /* optional fraction */
                       "." digit { digit }              /* digits after the mandatory point */
                       [ ( "E" | "e" )                  /* optional exponent part */
                         [ "+" | "-" ]                  /* optional sign within this */
                         digit { digit }                /* mandatory digit sequence */
                       ]
                     ] .
```

This will not match numbers like 123E45 or 123E-45. As an exercise, work out what you would need to allow such representations, as well as the ones we have seen so far.

The biggest traps that beginners fall into are as follows:

(a) The token names introduced on the left of the TOKENS production rules may not be used on the right side of the TOKENS rules in a direct or indirect way. Put another way, you cannot define one token in terms of another token, or recursively in terms of itself. So the following attempt to define a float number is invalid:

```
  TOKENS
    integer        = digit { digit } .               /* so far so good */
    floatnumber    = integer "." integer .           /* looks seductive! */
```

The only "names" you can use within the definition part, following the = sign, are names of character sets declared in the CHARACTERS section.

However, the token names invariably appear somewhere on the right sides of productions in the PRODUCTIONS section.

(b) All the token definitions must be given by uniquely distinguishable regular expressions. So you cannot write

```
  TOKENS
    firstName = ULetter { LLetter } .
    surname   = ULetter { LLetter } .
    longName  = Uletter LLetter { LLetter } .       /* at least two letters */
```

because all of these amount to describing the same patterns of characters. Note, however, that you could write (if you think it appropriate) something like

```
  TOKENS
    firstName = ULetter { LLetter } .
    surname   = ULetter { LLetter } "." .
    integer   =   digit { digits }
                | digit { hexDigit } "H" .
```

where the full stop at the end is sufficient to distinguish a `surname` (where it must appear) from a `firstName` (where it must not appear), or an implicitly decimal integer number from an implicitly hexadecimal number.

To sum up, the production rules in the TOKENS section are expressed in EBNF, but they do not have to be in LL(1) form (although they often are).

(c) Sometimes a token must be described in terms of a single element from a character set. This is a useful device - and in fact the only real way - for defining a terminal token that consists of a control character. By extension one might need to specify a simple sequence of awkward control characters:

```
  TOKENS
    EOL  = LF .
    CRLF = CR LF .
```

Note that there is one predefined token, denoted by the keyword `EOF`, that can be used in the PRODUCTIONS section to ensure that a grammar describes all the input up to the end of file (see below).

(d) A token definition specifies a complete token as a string of contiguous characters. A common mistake is to try to put too much into a token definition - for example

```
  TOKENS
    Header = "void main () {" .
```

This sort of construction should be specified in the PRODUCTIONS section in terms of discrete tokens:

```
  PRODUCTIONS
    Header = "void"     identifier     "("     ")"     "{" .
```

(e) In any event, spaces cannot appear within literal strings used to define tokens. If a token really needs to incorporate a space (very unusual!), this must be done using something like

```
  TOKENS
    Devil = "Pat" Space "Terry" .
```

Ignorable features and characters

Coco/R *optionally* allows one to specify that a set of characters is "ignorable", meaning that members of this set will be skipped over each time the scanner looks for the *next* token. The basic space character (SP, CHR(32)) is always in this set. Usually all the control characters - or at least the whitespace control characters - are also ignorable, meaning that they can be freely used to separate different tokens in the manner familiar to all Java programmers.

Because a character is ignorable does not mean that it cannot form an *internal* part of a token, but an ignorable character cannot be the *initial* character of a token.

Case-insensitive parsers/scanners can be constructed by using the keyword `IGNORECASE`, as shown below. If this is required, the `IGNORECASE` directive must appear before any others.

Examples

```
  COMPILER Example
    IGNORECASE                           /* case insensitive scanner */

    ...  /* CHARACTERS, TOKENS sections, followed by */

    IGNORE CHR(0) .. CHR(31) .           /* control characters */
    IGNORE CHR(9) + CHR(11) .. CHR(13) .  /* white space, but not LF */
    IGNORE CHR(9) .. CHR(13) .           /* all white space */
```

COMMENTS section

The *optional* COMMENTS section allows one to specify the format of comments that might be injected into the file for which a grammar is needed, but which will be ignored. That is, comments are treated like white space if they appear between tokens. Typically these comments are specified with a directive like

```
  COMMENTS FROM "(*" TO "*)" NESTED  /* Modula-2 style nested comments */
  COMMENTS FROM "/*" TO "*/"         /* Java style comments - 1 */
  COMMENTS FROM "//" TO EOL          /* Java style comments - 2 */
  COMMENTS FROM ";"  TO EOL          /* Assembler style comments */
```

In the last two examples `EOL` would have been defined under the TOKENS section as shown earlier.

One can have more than one comment format for a single grammar, but the tokens that delimit comments can be only one or two characters long.

PRODUCTIONS Section

All Cocol grammars must have a PRODUCTIONS section, and there must be a production for the `Goal` symbol, which is also used as the name of the grammar. The shortest possible Cocol specification (which, of course, does nothing useful) would thus be

```
  COMPILER Goal
  PRODUCTIONS
    Goal = .
  END Goal.
```

Within the PRODUCTIONS section there must be one (and only one) defining production for each non-terminal. This production can, of course, be defined to allow alternatives. The productions can appear in any order.

Cocol can only be used for context-free grammars, so that the left side of each production must always be a single non-terminal. The right side *can* include

- Terminals represented directly by strings - for example `"void"`
- The names of terminal tokens named under the TOKENS section - for example `Number`
- The names of other non-terminals - for example `Expression`
- The meta-characters | { } [ ] and ( ) .

The right side *cannot* include

- The names of character sets as defined in the CHARACTERS section, however tempting it might be to do so. Use an intermediate token definition, as illustrated previously.

A typical production might be

```
  Factor = number | [ "sqrt" ] "(" Expression ")" .
```

Productions may be recursive, but left recursive grammars are intrinsically non-LL(1) and thus rarely of any use. In general Coco/R only generates usable parsers if the PRODUCTIONS section defines an LL(1) grammar. There are a few exceptions to this - for example the "dangling else" construction is non-LL(1), but a Coco/R generated parser does the sensible thing.

Empty alternatives can be defined using square brackets (the recommended way), or by using a | followed by nothing, for example:

```
  IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ] .
  IfStatement = "if" "(" Condition ")" Statement ( "else" Statement | ) .
```

Be careful of the precedence rules - alternation | is "weaker" than concatenation, so that

```
    A = B C | D E .
```

means

```
    A = ( B C )       |       ( D E ) .
```

and not

```
    A = B       ( C | D )       E .
```

This is a very easy mistake to make, and as with forgetting to put braces round blocks of statements in Java programs, it can lead to very-hard-to-find bugs. The production below is acceptable to Coco/R, but is (hopefully obviously) *wrong*:

```
  IfStatement = "if" "(" Condition ")" Statement "else" Statement | .
```

A production like

```
  Goal = { Something } EOF .
```

is often (but not always) preferable to the simpler, less explicit

```
  Goal = { Something } .
```

The first form will signal an error if the sequence of `Somethings` does not reach the end of file, the second form might just stop prematurely and leave the user wondering what happened.

Be careful to avoid introducing optional parts in too many ways. This sometimes happens accidently, often indirectly. For example, do not fall into the trap of writing productions like

```
  Complete      = [ { Part1 } { Part2 } ] .
  Something     = [ [ Inner ] ] .
  LotsOfOptions = { [ First ] [ Second ] [ Third ] } .
  WholePart     = [ Part1 Part2 ] .
  Part1         = { Something } .
  Part2         = [ Part3 ] .
```

Cocol Pragmas

A Cocol grammar can include pragmas - generally placed right near the start. These start with a $, followed by a few key letters, for example $CNF. Letters of interest are

- C - generate a driver program as well as the scanner and parser
- N - generate names for the tokens (like `pointSym` or `commaSym`) rather than cryptic numbers
- F - list the FIRST and FOLLOW sets for each non-terminal (in the file `listing.txt`).
- T - test the grammar but do not generate any scanner, parser or driver

---

Home © P.D. Terry